TP15 : Concurrence

Objectifs du TP :

- Manipuler quelques fonctions du module Thread en Ocaml et de la bibliothèque pthread en C.
- Utiliser un mutex ou un sémaphore dans un contexte simple.
- Observer des comportements théorisés en cours : entrelacement, non atomicité, attente active...
- Implémenter l'algorithme de Petersen.
- Accélérer un calcul en utilisant le parallélisme.

Ce sujet est constitué d'exercices indépendants à aborder en C ou en Ocaml. Il est précédé de précisions quant aux prototypes des fonctions des bibliothèques pthread et semaphore et à la façon de les utiliser.

Cours *Fils et mutex en C*

Création d'un fil

Le prototype de la fonction permettant d'initialiser et lancer un thread est :

```
int pthread_create(pthread_t* thread, pthread_attr_t* attr,
                   void* f(void*), void* arg)
```

Le premier argument est un pointeur vers un objet de type pthread_t préalablement créé. Le deuxième consiste en des options : on utilisera toujours NULL. Les deux derniers arguments spécifient la fonction que le fil doit exécuter et son argument.

La fonction pthread_create renvoie un entier indiquant si le lancement du fil s'est déroulé correctement (l'entier renvoyé est alors 0) ou non. Le nouveau fil commence immédiatement à s'éxécuter de manière concurrente avec le fil qui a réalisé l'appel à pthread_create.

Le prototype de la fonction à exécuter est obligatoirement :

```
void* mafonction(void* arg);
```

C'est une fonction qui prend en entrée un pointeur "générique" et renvoie un pointeur générique. En pratique, il faudra transtyper le pointeur en entrée afin de pouvoir récupérer la valeur pointée. Par exemple, si la fonction qu'on souhaite faire exécuter à un fil consiste à afficher n fois "toto" où n est un entier argument de la fonction, on écrira :

```
void* affiche(void* n)
{
    //On transtype pour obtenir un pointeur du bon type.
    int* n_transtype = (int*)n;
    //On peut maintenant déréférencer.
    int borne = *n_transtype;
    //Puis utiliser l'argument comme souhaité.
    for (i = 0; i < borne ; i++)
    {
        printf("toto");
    }
}
```

Si on veut ensuite un fil qui exécute la fonction affiche avec $n = 10$, on pourra écrire dans le main :

```
int N = 10;
pthread_t p1;
pthread_create(&p1,NULL,affiche,&N);
```

Attente de fin d'exécution

Si un fil d'exécution en lance un autre, le fil "interne" sera automatiquement interrompu dès que le fil "englobant" aura terminé son exécution. Pour que le fil englobant laisse le fil interne terminer, il faut lui spécifier de l'attendre à l'aide de la fonction de prototype :

```
int pthread_join(pthread_t thread, void **retval);
```

Le premier argument est le nom du fil à attendre, le deuxième est un argument qui permet de récupérer la valeur calculée par la fonction exécutée par le fil interne. Pour nous, le deuxième argument sera toujours fixé à NULL.

L'entier renvoyé par pthread_join est 0 si l'appel à cette fonction est un succès et autre chose sinon : on l'ignorera. L'effet de cette fonction est de bloquer le fil dans lequel l'appel à pthread_join est fait jusqu'à ce que le fil dont l'identifiant est donné à pthread_join ait terminé son exécution.

Mutex

Les fonctions utiles à connaître sur les mutex sont les suivantes :

```
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

La première permet d'initialiser un mutex : elle prend en entrée un pointeur vers le mutex à initialiser (de type pthread_mutex_t). Le deuxième argument de cette fonction correspond à des options : pour nous, ce sera toujours NULL.

La fonction pthread_mutex_lock permet de verrouiller le mutex dont l'adresse est donnée en argument ; la fonction pthread_mutex_unlock fonctionne de manière similaire. Si un fil qui n'a pas verrouillé le mutex appelle pthread_mutex_unlock, le comportement du code est indéfini.

La fonction pthread_mutex_destroy permet de libérer toutes les ressources éventuellement associées au mutex pointé par son argument. Une fois détruit, on ne peut plus utiliser le mutex. Appeler pthread_mutex_destroy sur un mutex verrouillé résulte en un comportement indéfini.

Sémaphores

Pour utiliser un sémaphore en C, il faut inclure l'en-tête semaphore.h. Elle nécessite d'utiliser l'option -pthread à la compilation. Les fonctions que l'on utilisera sur les sémaphores sont les suivantes :

```
int sem_init(sem_t* sem, int pshared, unsigned int value);
int sem_wait(sem_t* sem);
int sem_post(sem_t* sem);
int sem_destroy(sem_t *sem);
```

Toutes renvoient 0 si l'opération décrite ci-dessous s'est déroulée correctement et autre chose sinon ; on ignorera les valeurs de retour de ces fonctions. Par ailleurs :

- La fonction sem_init permet d'initialiser un sémaphore : elle prend en entrée un pointeur vers le sémaphore à initialiser, un entier indiquant si le sémaphore doit ou non être partagé par plusieurs fil - si oui, cette valeur doit être 0 - et la valeur initiale du compteur.
- La fonction sem_wait prend en entrée un pointeur sur un sémaphore qu'elle décrémente si c'est possible. Si ce n'est pas possible, le fil qui a appelé sem_wait est mis en attente (non active).
- La fonction sem_post prend en entrée un pointeur sur un sémaphore et l'incrémente. Si le compteur du sémaphore devient strictement positif, un fil bloqué dans un appel à sem_wait est réveillé et peut ainsi décrémenter le sémaphore et poursuivre son exécution.
- La fonction sem_destroy remplit la même fonction que pthread_mutex_destroy mais sur les sémaphores plutôt que les mutex.

Cours *Compilation en C et Ocaml lors de l'utilisation de fils*

En C :

- Il faut inclure l'en-tête pthread.h et compiler avec l'option -pthread.
- Dans le cas où on utilise des sémaphores, il faut en plus l'en-tête semaphore.h.
- Pour détecter les erreurs de synchronisation à l'exécution, compiler avec l'option -fsanitize=thread. Cette option est incompatible avec -fsanitize=address.

En Ocaml :

- Si vous compilez votre code source, la commande est

```
ocamlc -I +threads unix.cma threads.cma nom_fichier.ml
```

- Pour utiliser le module Thread directement dans l'interpréteur, les premières lignes de votre code source doivent être :

```
#directory "+threads";;
#load "unix.cma";;
#load "threads.cma";;
```

Exercice 1 *Entrelacement*

Cet exercice est à traiter en Ocaml.

1. Ecrire une fonction affichage prenant en entrée un entier i et produisant les affichages suivants :
   - "Le fil i a démarré".
   - "Le fil i a atteint d" pour tout $d \in [\![1, n \times p]\!]$ tel que d est multiple de n.

   Les entiers n et p seront définis via deux variables globales.
2. Ecrire une fonction lançant 3 fils d'exécution : le fil numéro i exécutera la fonction affichage avec i comme argument.
3. Tester cette fonction avec $n = 10000000$ et $p = 3$ et observer le non déterminisme des affichages.

Exercice 2 *Non atomicité de l'incrément*

Cet exercice est à traiter en C.

1. Ecrire un programme compteur_faux utilisant deux fils pour incrémenter un compteur partagé initialement nul et non protégé par un mutex de 200000 unités en tout.
2. Lancer plusieurs fois le programme précédent, observer et expliquer le résultat.
3. Ecrire un programme compteur corrigeant le programme précédent à l'aide d'un mutex.

Exercice 3 *Un code faux*

Cet exercice est à traiter en C. On considère le code suivant (fichier faux.c) :

```c
#include <stdio.h>
#include <pthread.h>

#define NB_THREADS 2

void *f(void *arg)
{
    int index = *(int*)arg;
    for (int i = 0; i < 10; i++)
    {
        printf("Thread %d : %d\n", index, i);
    }
    return NULL;
}
int main()
{
    pthread_t threads[NB_THREADS];
    printf("Before creating the threads.\n");
    for (int i = 0; i < NB_THREADS; i++)
    {
        int thread_index = i;
        pthread_create(&threads[i], NULL, f, &thread_index);
    }
    printf("While the other threads are running.\n");
    for (int i = 0; i < NB_THREADS; i++)
    {
        pthread_join(threads[i], NULL);
    }
    printf("After the other threads have stopped running.\n");
    return 0;
}
```

1. Compiler et exécuter ce code. Que se passe-t-il ?
2. Si ce n'était pas déjà fait, compiler en utilisant l'option -fsanitize=thread. Etudier le message obtenu.
3. Expliquer le comportement de ce code et où se situe le problème.
4. Proposer une version corrigée de ce code.

Exercice 4 *Algorithme de Petersen*

Cet exercice est à traiter en C. Son objectif est de traduire l'algorithme de Petersen pour l'implémentation d'un mutex vu en cours. Pour ce faire, on se dote de la structure suivante :

```
struct petersen {
    bool want[2];
    int turn;
};
typedef struct petersen petersen;
```

Le champ turn indique le numéro du fil (0 ou 1) qui a la priorité.

1. Ecrire une fonction petersen* init(void) qui renvoie un pointeur vers un verrou dont le champ want est un tableau contenant false et le champ turn est initialisé à 0.
2. Ecrire une fonction void lock(petersen* m, int thread) verrouillant le mutex pris en entrée selon l'algorithme de Petersen. L'argument thread est le numéro du fil qui effectue l'appel.
3. Ecrire de même une fonction void unlock(petersen* m, int thread).
4. Ecrire un programme parallèle simple où deux fils incrémentent 1000000 de fois de manière concurrente un compteur protégé par le verrou qu'on vient de définir.
5. Tester ce programme avec différentes options de compilation (avec -O0 pour n'avoir aucune optimisation lors de la compilation, avec -O1, avec et sans -fsanitize=thread). Que constate-t-on ?

Exercice 5 *Intérêt de la programmation parallèle*

Cet exercice est à traiter en C : il s'agit de compléter le fichier somme.c.

Son objectif est de calculer la somme des éléments d'un tableau de flottants en utilisant plusieurs fils d'exécution afin d'accélérer ledit calcul. On procède naïvement en définissant une variable globale SUM dans laquelle on accumule le résultat. Dans les versions parallèles, on s'aidera du type suivant : il représente une partie d'un tableau. Plus précisément, arr pointe vers la première case de la partie étudiée et len indique la longueur du morceau de tableau étudié.

```
struct thread_args {
    double* arr;
    int len;
};
typedef struct thread_args thread_args;
```

1. Compléter la fonction partial_sum_1 pour quelle ajoute les éléments de la partie du tableau spécifiée par son argument à la variable SUM en faisant comme si il n'y avait aucun problème du à la concurrence (sans mutex donc).
2. Ecrire de même une fonction partial_sum_2 : elle devra ajouter les éléments un à un en protégeant chacun des ajouts à l'aide d'un mutex.
3. Faire de même avec partial_sum_3 : elle devra accumuler sa propre somme partielle dans une variable locale puis ajouter son résultat partiel à SUM en protégeant l'ajout.
4. Compléter le main pour obtenir un programme qui prend en entrée un nombre de fils et une longueur de tableau, génère un tableau avec la fonction fournie et en calcule la somme :
    - Une fois de manière naïve avec une simple boucle.

- Avec chacune des fonctions précédentes.

5. Comparer les différentes versions de partial_sum (en modifiant éventuellement le main pour faire des affichages supplémentaires) sur les critères suivants : correction de la valeur renvoyée et temps de calcul (on pourra utiliser les fonctions déclarées dans le fichier clock.h). Commenter.

Exercice 6 *Sémaphores en C*

Cet exercice est à traiter en C. Il consiste à compléter le fichier pc.c. On considère un problème producteur-consommateur avec un buffer de taille $n \in \mathbb{N}^\star$. Pour éviter une attente active d'un fil (qu'on observera dans l'exercice suivant), on utilise deux sémaphores ayant les sémantiques suivantes :

- Le sémaphore full indique le nombre de places restantes dans le buffer. Il vaut donc 0 lorsque le buffer est plein ; auquel cas les producteurs ne doivent plus pouvoir y écrire.
- Le sémaphore empty indique le nombre d'éléments présents dans le buffer. Il vaut donc 0 lorsque le buffer est vide ; auquel cas, les consommateurs ne doivent plus pouvoir y consommer.

1. Comprendre le fonctionnement de la fonction produce et écrire une fonction analogue consume qui consomme 20 entiers en affichant Consumption i pour tout $i \in [\![0, 19]\!]$.
2. Dans le main, faire interagir un fil consommateur exécutant consume et un fil producteur exécutant produce dans le cas où le buffer est de taille 5.
3. Exécuter votre programme et observer les affichages : est-ce cohérent ?

Exercice 7 *Producteurs et consommateurs en Ocaml*

Cet exercice est à traiter en Ocaml. On reprend le problème des producteurs-consommateurs étudié en cours. On suppose que le buffer est de taille non bornée. Il sera modélisé par le type suivant :

```
type 'a buffer = {data : 'a Queue.t;
                  mutable count : int;
                  lock : Mutex.t;}
```

Le champ data est une file contenant les éléments dans le buffer, le champ count indique combien il y a d'éléments dans cette file et le champ lock est un mutex que les différents fils seront suceptibles de verrouiller et déverrouiller pour pouvoir écrire ou consommer dans le buffer. Dans les questions 1 à 6, on n'utilisera pas de sémaphores.

1. Ecrire une fonction create_buffer : unit -> 'a buffer permettant de créer un buffer vide.
2. Ecrire une fonction produce : 'a buffer -> 'a -> unit permettant à un fil d'ajouter un élément dans le buffer de manière sûre.
3. Ecrire une fonction consume : 'a buffer ->'a permettant à un fil de lire dans un buffer de manière sûre. L'élément renvoyé doit être l'élément en tête de file. Si le buffer est vide, le fil attendra activement qu'un élément soit produit.
4. Ecrire une fonction producer : (int*int) buffer -> int -> int -> unit tel que producer b n id écrive dans le buffer b les couples (id, i) pour tout i entre 0 et n-1 avec un délai aléatoire entre 0 et 2 secondes entre chaque écriture. On pourra à cette fin utiliser Unix.sleepf.
5. Ecrire une fonction consumer : (int*int) buffer -> int -> unit tel que consumer b n lise n éléments dans le buffer b, les stocke dans une liste et les affiche dans l'ordre dans lequel ils ont été lus.

6. A l'aide des fonctions précédentes, faire écrire 10 éléments chacun à deux fils producteurs dans un buffer alors qu'un fil consommateur tente de lire 20 éléments dans ce même buffer. Tester ce code et observer la consommation de votre machine pendant cette exécution. Est-ce satisfaisant ?

7. (bonus) Corriger ce problème à l'aide d'un sémaphore. *Il est possible que la version d'Ocaml sur vos machines ne soit pas suffisamment récente pour autoriser l'utilisation de sémaphores en pratique. Que cela ne vous empêche pas de réfléchir au code !*